

Memory layout in GPU implementation of lattice Boltzmann method for sparse 3D geometries

Tadeusz Tomczak^{a,*}, Roman G. Szafran^b

^aWrocław University of Technology, Faculty of Electronics, Chair of Computer Engineering, Janiszewskiego 11/17, 50-370 Wrocław, Poland, e-mail: tadeusz.tomczak@pwr.wroc.pl

^bWrocław University of Technology, Faculty of Chemistry, Department of Chemical Engineering, C.K. Norwida 4/6, 50-373 Wrocław, Poland, e-mail: roman.szafran@pwr.wroc.pl

Abstract

We describe a high-performance implementation of the lattice Boltzmann method (LBM) for sparse 3D geometries on graphic processors (GPU). The main contribution of this work is a data layout that allows to minimise the number of redundant memory transactions during the propagation step of LBM. We show that by using a uniform mesh of small three-dimensional tiles and a careful data placement it is possible to utilise more than 70% of maximum theoretical GPU memory bandwidth for D3Q19 lattice and double precision numbers. The performance of our implementation is thoroughly examined and compared with other GPU implementations of LBM. The proposed method performs the best for sparse geometries with good spatial locality.

Keywords: lattice Boltzmann method, LBM, GPU, CUDA

1. Introduction

The lattice Boltzmann method (LBM) is a new versatile and highly parallel approach where the discrete Boltzmann transport equation is solved in the velocity or moment \mathbf{R}^n space to obtain the time-dependent fluid velocity distributions. Due to a trivial LBM parallelization many researchers have explored the area of its implementations on graphics processors (graphic processing units, GPU). A thorough review is presented in [1]. A few typical optimisation techniques may be mentioned: combining all computations into the single kernel to minimise memory bandwidth usage, use of structure of arrays (SoA) instead of array of structures (AoS) data types to avoid uncoalesced memory transactions, an adaptation of all computations to the single precision floating point format to minimise a register pressure and maximise occupancy, use of two copies of data to avoid race conditions, and many others.

However, the above optimisations concern dense geometries. The number of GPU implementations for sparse geometries (with many solid nodes) is much lower and they offer significantly worse performance. This can hinder the usage of GPU LBM implementations in many areas where dense geometries are not applicable. Typical examples may be biomedical or porous media simulations [2].

Sparse geometries handling requires some form of storing information about the placement of non-solid nodes. This information can not only increase memory usage, but

also generate irregular memory access patterns that can significantly decrease performance.

In general, two indirect addressing solutions are used in GPU implementations of LBM for sparse geometries: the Connectivity Matrix (CM), shown in [3] with highly optimised version in [2], and the Fluid Index Array (FIA), presented in [4]. The connectivity matrix contains pointers to the neighbour node for each propagated f_i function. The fluid index array is a kind of "bitmap" that contains -1 for every solid node in geometry or the pointer to non-solid node data. Both solutions bring an overhead in memory usage, additionally the FIA implementation results in a low utilisation of memory bandwidth due to the irregular data access pattern.

In this work we present the GPU implementation of LBM for sparse geometries where the information about geometry sparsity is stored using the uniform grid of three-dimensional cubic tiles. Due to the fixed tile size, we were able to arrange data inside a tile to minimise memory traffic. For the proposed data layout, we show both a detailed theoretical analysis and results of performance measurements for dense and sparse geometries. Additionally, we analyse how the fixed tile size affects the overall performance. Provided that the tiles contain a low number of solid nodes, the performance of our implementation remains high regardless of the geometry sparsity.

The structure of the paper is as follows. In Section 2, we briefly introduce the LBM method and define basic concepts in GPU programming. Section 3 contains the description of our implementation with the detailed analysis of introduced overheads. The performance comparison

*Corresponding author.

E-mail address: tadeusz.tomczak@pwr.wroc.pl

with existing implementations and a verification procedure are presented in Section 4. Section 5 contains conclusions.

2. Background

2.1. Graphic Processing Unit

Today’s GPUs are in fact mass-produced, easily programmable versatile variant of vector coprocessors. Compared with universal Central Processing Unit (CPU), they offer more computational power (due to numerous units) and a higher memory bandwidth at the cost of a diminished cache memory and a low performance of a single computational unit. Two of the most popular frameworks for GPU programming are OpenCL and CUDA (in this work we use the latter). Selected parameters of CUDA capable GPU generations are shown in Table 1. It can be seen that the successive generations offer not only a higher memory bandwidth, but also their computational power for double precision increases faster than the memory bandwidth.

Table 1: Approximate parameters of CUDA capable GPUs. We consider only models with maximum double precision performance dedicated to the high performance computing. The last row contains parameters of a high-end general purpose CPU.

Architecture	Launch year	Bandwidth [GB/s]	DP FLOP/B
Tesla	2008	~ 100	~ 0.8
Fermi	2009	~ 150	~ 3.5
Kepler	2012	~ 250	~ 5.5
Pascal	2016	~ 720	~ 7.4
Broadwell	2016	~ 80	~ 8

In the CUDA programming model [5] a complete coprocessor (*device*) contains a GPU with a separate memory. The device is controlled by a *host* computer that launches on the GPU many instances (*threads*) of a GPU code (*kernel*). Threads are arranged in an explicitly defined multi-dimensional structure: 3D *grid* of 3D thread *blocks*.

From the hardware point of view, a CUDA GPU contains a multi-channel DDR memory controller, optional L2 cache memory and from one up to over a dozen of *multiprocessors*. Each multiprocessor has a number of processing units, a register file, an amount of fast *shared* memory (used as a software controlled scratchpad), cache memories and a number of instruction decoders. One instruction decoder is common for many computational units, thus threads are run in groups called *warps*. The situation, when threads from the same warp need to follow different execution paths, is called thread *divergence* and causes a performance penalty.

Threads are assigned per multiprocessor in complete blocks. Usually, the number of blocks concurrently run

on the multiprocessor is limited by the usage of registers and shared memory, lowering the multiprocessor *occupancy*. High occupancy and/or high instruction level parallelism (ILP) within single threads allow to achieve a high performance through masking of instruction latencies by hardware task switching.

GPU memory hierarchy contains the off-chip DDR DRAM *global* memory (limited to 1–32 GB depending on GPU), up to two levels of on-chip cache memory (there are separate caches for read-only access), shared memory and registers. A single transaction with the global memory requires a few hundreds of GPU clock cycles, thus memory transactions for threads from the same warp can be *coalesced* into one if defined restrictions are met. Additionally, limited software control over a cache usage is available.

2.2. Lattice-Boltzmann method

In the LBM, as in the conventional CFD, the geometry, initial and boundary conditions must be specified to solve the initial-value problem. The computational domain is uniformly partitioned and computational nodes are placed in vertices of adjacent cells, giving the lattice. In this study, the D3Q19 lattice structure is used (D3 is the space dimension, Q19 is the lattice links number).

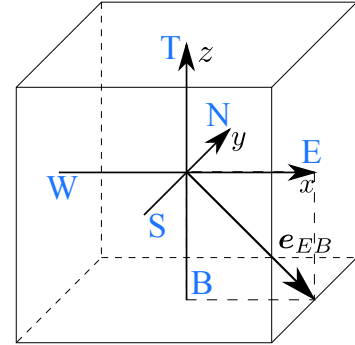


Figure 1: The convention of lattice directions naming. A single unit vector $\mathbf{e}_{EB} = \langle 1, 0, -1 \rangle$ is marked as an example.

Let f_i represent the probability distribution function along the i lattice direction, δ_x and δ_t are the lattice spacing and the lattice time step (usually assumed to be equal 1), $\frac{\delta_x}{\delta_t}$ is the lattice speed (also usually assumed to be equal 1 lu), \mathbf{e}_i is the unit vector along the i lattice direction, and $\mathbf{c}_i = \frac{\delta_x}{\delta_t} \mathbf{e}_i$ is the lattice velocity vector in the velocity space along the i lattice direction. In this work we use the lattice directions naming scheme shown in Fig. 1. The core of the lattice Boltzmann method is the discretized in velocity \mathbf{R}^n space form of the Boltzmann transport equation that can be written for each i of q directions (lattice linkages) in velocity space as follows:

$$\frac{\partial f_i}{\partial t} + \mathbf{c}_i \cdot \nabla f_i = \Omega_i, \quad (1)$$

where Ω_i is the collision operator.

The collision operator can be approximated with the most popular Bhatnagar-Gross-Krook model [6],

$$\Omega_{BGK} = -\frac{1}{\tau} (f_i - f_i^{eq}), \quad (2)$$

where τ is the relaxation time in LB units related to the lattice fluid viscosity and f_i^{eq} is the equilibrium distribution function along the i lattice direction given by the following formula for the quasi-compressible model:

$$f_i^{eq} = \omega_i \rho \left(1 + \frac{\mathbf{c}_i \cdot \mathbf{u}}{c_s^2} + \frac{(\mathbf{c}_i \cdot \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}^2}{2c_s^2} \right) \quad (3)$$

and for the incompressible model [7]:

$$f_i^{eq} = \omega_i \left(\rho + \frac{\mathbf{c}_i \cdot \mathbf{u}}{c_s^2} + \frac{(\mathbf{c}_i \cdot \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}^2}{2c_s^2} \right) \quad (4)$$

where c_s is the lattice speed of sound that is a lattice constant; \mathbf{u} is the macroscopic fluid velocity vector expressed in LB units; $\rho = \sum_i f_i$ is a fluid density expressed in LB units, and ω_i is a weighting scalar for the i lattice direction. The macroscopic velocity for the quasi-compressible model can be determined from

$$\mathbf{u} = \frac{1}{\rho} \sum_i \mathbf{c}_i f_i \quad (5)$$

and for the incompressible model from [8]

$$\mathbf{u} = \sum_i \mathbf{c}_i f_i. \quad (6)$$

Integrating Eqn. (1) from t to $t + \delta_t$ along the i lattice direction and assuming that the collision term is constant during the interval, we can obtain discretized in time form of BGK-LBM equation

$$\underbrace{f_i(\mathbf{r} + \mathbf{c}_i \delta_t, t + \delta_t) - f_i(\mathbf{r}, t)}_{\text{Streaming}} = \underbrace{\frac{\delta_t}{\tau} [f_i^{eq}(\mathbf{r}, t) - f_i(\mathbf{r}, t)]}_{\text{Collision}}, \quad (7)$$

where \mathbf{r} is a position vector in the velocity space. The term on the left hand side is known as the streaming step, the latter represents the collision step. These two steps are repeated sequentially during the simulation giving velocity, density and pressure distributions at each time step.

The BGK-LBM uses the single relaxation time to characterise the collision effects. However, physically, these rates should be different during collision processes. To overcome this limitation, a collision matrix with different eigenvalues or multiple relaxation times can be used. The LBM with an MRT collision operator can be expressed as [9]:

$$\underbrace{f_i(\mathbf{r} + \mathbf{c}_i \delta_t, t + \delta_t) - f_i(\mathbf{r}, t)}_{\text{Streaming}} = \underbrace{\mathbf{A} [\mathbf{f}^{eq} - \mathbf{f}]}_{\text{Collision}}, \quad (8)$$

where \mathbf{A} is the collision matrix. The MRT-LBM model has been attracting more attention recently due to the higher stability than that of the BGK-LBM model. In our study, we use both above mentioned collision models in incompressible and quasi-compressible versions to compare their efficiency.

2.3. LBM performance limits

To compare different implementations, we employ a widely used, simple computational complexity model that allows for computing the minimum memory usage, the memory bandwidth and the computational performance. Subsequently, we compare the real results to this theoretical limit.

Since we are interested in peak performance, in the considerations that follow below we ignore boundary nodes. We also assume that the computational unit (CPU or GPU) does computations for a single node in a single LBM iteration (streaming and collision) in three stages: data read from the main memory, computations, data write to the main memory (the data for a single lattice node is buffered in registers and in the internal memory, e.g. shared/cache memory). It was shown in [10] that it is possible to do even a few time steps per one data read/write from the main memory, but according to [11] this method was difficult to apply to more complex boundary condition models.

We assume that for each lattice node the only values stored in the main memory are f_i functions and an additional node type identifier. Since the previous work has shown that usually LBM implementations are memory bound, the other values (e.g. velocity or density) can be computed internally, and dropped after use without transferring them from/to the main memory.

Let q denote the number of functions f_i , n_d denote the number of bytes for storing a single f_i value (e.g. 4 B for a single precision floating point, 8 B for double precision), and n_t denote the number of bytes used to store a node type (we assume that at least one byte is necessary, although for simple cases only a few bits can be enough). Thus, the minimum number of memory bytes needed for single node datum is

$$M_{node} = q \cdot n_d + n_t \quad [B]. \quad (9)$$

For a single LBM iteration, the new f_i values are computed based on the node type and the previous f_i values. The newly computed f_i values are stored in the memory. The minimum number of bytes transferred per one LBM iteration for a single node is then

$$B_{node} = 2 \cdot q \cdot n_d + n_t \quad [B]. \quad (10)$$

Formulae defining the number of floating point operations (FLOP, not to be confused with floating point operations per second, FLOPS) are much more complex to determine. A simple count of the operations resulting from

Table 2: Computational complexity of LBM operations for single fluid node, D3Q19 lattice and double precision values. First four rows show complexity for complete collision and v, ρ computations. The last three rows shows complexity of separate v, ρ computations - for quasi-compressible fluid model the computation of v, ρ requires 3 additional divisions. Separate collision complexity can be calculated by subtracting complexity of v, ρ computations from values from the first four rows. FMA (fused multiply-add) counts as two floating point operations. FSETP and FREC denote GPU instructions for floating point condition testing and reciprocal computing. FLOP/B ratio is calculated assuming 306 bytes transferred per node (see Eqn. 10).

Operation	FADD	FMUL	FMA	FSETP	FREC	# instr.	FLOP	FLOP/B
BGK incompressible	65	21	109	–	–	195	304	0,99
BGK quasi-compressible	65	39	166	21	6	297	463	1,51
MRT incompressible	324	40	329	–	–	693	1022	3,34
MRT quasi-compressible	323	43	386	21	6	780	1165	3,81
v, ρ incompressible	49	–	–	–	–	49	49	
v, ρ quasi-compressible	49	15	57	21	6	148	205	
FPU division	–	5	19	7	2	33	52	

a naive implementation of equations (7) and (8) gives numbers significantly larger than in real implementations. Actually, many operations described by equations (7) and (8) can be skipped (e.g. multiplications by $-1, 0$ or 1) or used a few times (e.g. partial sums of f_i). Some of these optimisations can be detected very early (i.e. multiplications by e_i constants equal to $-1, 0, 1$), but others are unpredictable since they are applied during the compilation and their use depends on many factors (optimisation level, registers usage constraints during compilation, machine capabilities - a number of registers etc.). Thus, for numbers of computational operations we show only specific values obtained by a disassembling of a GPU binary code using *nvdiasm* utility (we count only arithmetic operations). The results are shown in Table 2.

The number of FLOP for our LBM implementation consists of two parts: computation of v, ρ and collision. The complexity of v, ρ computations depends only on the fluid model. These computations are implemented as a simple series of additions and optional divisions for the quasi-compressible model. The computational complexity of collision depends practically almost only on collision model. The full cost of the quasi-compressible model is then built-in into computations of v, ρ , where additional divisions are needed for each velocity component, and into the code responsible for boundary nodes handling (not shown in Table 2). Due to a different ratio of division to collision complexity, the computational complexity of the quasi-compressible model implementation is from 50% for BGK to 14% for MRT higher than for the incompressible model.

The comparison of FLOP/B ratios from Table 2 with specifications of high-performance GPUs (see Table 1) shows that LBM implementations are usually bandwidth bound. This is especially true for the newest CUDA capable architectures, for which the FLOP/B ratio usually gradually increases. The high performance LBM implementations for GPU should then focus on memory bandwidth optimisations. Notice that this is also true for general purpose CPUs, provided that their computational

power is fully utilised using vector instructions.

3. Implementation

3.1. Tiling overview

In our implementation, the whole geometry is covered by the uniform mesh of cubic tiles, where each tile contains a^3 nodes. If a geometry size is not divisible by a , then the geometry is extended with solid nodes. The tiling is implemented by the host code and done once at the geometry load. We use a very simple tiling algorithm: first, the geometry is covered by the uniform mesh of tiles starting at node $(0,0,0)$, and next, the tiles containing only solid nodes are removed.

In general, the tile size could be arbitrary but, as we show below, it can have a significant impact on performance due to a low tile utilisation if a tile is too large. Additionally, the number of nodes within a tile should be a multiple of the GPU warp size to avoid a low hardware utilisation. In our implementation we have chosen $a = 4$.

Fig. 2 shows thread assignment to nodes within a tile. We use thread blocks with dimension $\langle 4, 4, 4 \rangle$ so both coordinates of threads and of nodes in the tile are the same. This mapping gives two warps per tile: the first warp operates on all nodes with $z \in \{0, 1\}$, the second warp on nodes with $z \in \{2, 3\}$.

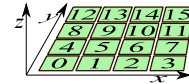


Figure 2: Thread assignment to nodes inside a tile for $threadIdx.z = 0$. Numbers denote linear thread index calculated as $threadIdx.x + 4 \cdot threadIdx.y + 16 \cdot threadIdx.z$. Full thread block consists of 4 such planes along z axis making 64 threads numbered from 0 to 63.

As many other LBM implementations, we use two copies of f_i values (denoted as f_i and f'_i) to avoid race conditions.

Thus, the tiles can be processed independently and in any order during a single LBM iteration.

Though the tiling is not a very novel technique (it is described even in NVIDIA tutorials as a method improving the performance of dense matrix multiplication), to the best of our knowledge this is the first attempt to use it for a GPU implementation of LBM for sparse geometries. Existing GPU LBM implementations for sparse geometries [3], [4], [2] focus on indirect addressing, what does not guarantee proper memory layout and requires additional data increasing memory and bandwidth usage. Below we will show that due to the regular memory layout the tiling offers a higher performance provided that the tiles contain a low number of solid nodes.

3.2. Tiling memory layout

Since the LBM implementations are usually bandwidth-bound on high-performance GPUs, the proper memory layout resulting in good memory bandwidth utilisation is crucial to achieving high performance. Thanks to the fixed tile size in our implementation we are able to fully control data placement for nodes processed in threads from the same warp. This gives us a possibility to maximise the number of coalesced memory transactions, what results in a very high utilisation of GPU memory bandwidth (comparable to values reported for dense geometries).

The general memory layout used in our implementation is shown in Fig. 3. All floating values for all nodes are stored in a single, continuous *allValues* array, what allows us to control data placement in GPU memory. In addition to the *allValues* array, we also use additional data arrays, e.g.: *nodeType* array where an information about LB nodes is stored, and *tileMap* array with tile bitmap. Data layout in the *nodeType* array is similar to *allValues* but for each tile only one block of data is stored. The *tileMap* array is a simple three-dimensional array stored in row order.

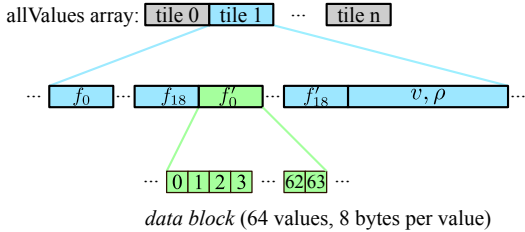


Figure 3: Memory layout of array with all values (f_i, v, ρ) for nodes within tiles. We use two copies of f_i values. For different f_i we use different assignment of node coordinates to position in 64-element *data block*.

A single value (e.g. f_0) from all nodes within the same tile forms a *data block* shown in Fig. 3. Data layout in *allValues* array guarantees that each data block (64 double precision values) is properly aligned and can be transferred using sixteen 32-byte wide memory transactions. The only problem is such data arrangement that

avoids uncoalesced/redundant transactions during propagation. To address this problem, each data block uses one of the three proposed data arrangements. Each data arrangement is optimised for different access patterns during propagation (see Fig. 4, 5 and 6).

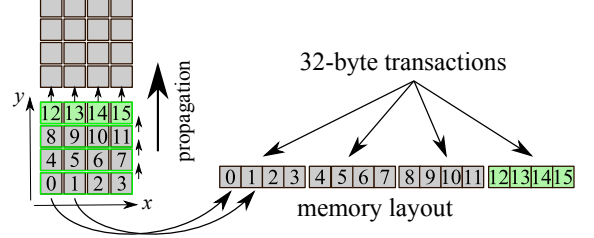


Figure 4: Propagation in north (N) direction for XYZ memory layout and double precision f_i values (8 bytes per value). Numbers denote linear memory offset of f_i value for corresponding node (for example: f_i for node at $x = 0, y = 1$ is placed at offset 4). Four consecutive double precision f_i values (one row) form a single 32-byte transaction. During propagation only four 32-byte memory transactions (reads) are required: one from the neighbour tile (read of nodes 12 to 15) and three from the current tile (read of nodes 0 to 11).

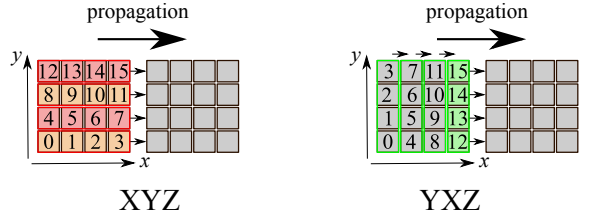


Figure 5: Propagation in east (E) direction for two memory layouts and double precision f_i values. For XYZ layout the four 32-byte wide memory transactions (marked with red rectangles) are needed to update nodes from the neighbour tile (nodes with indices 3,7,11,15). YXZ layout results in only one 32-byte transaction for nodes from the neighbour tile, because now these nodes have close indices (12,13,14,15). Additionally, during the propagation inside the tile only three (instead of four) 32-byte read transactions are done for nodes with indices 0 to 11. The fourth transaction is done from the neighbour tile.

Let $x_t, y_t, z_t \in \{0, 1, 2, 3\}$ denote node coordinates inside a tile. In all below considerations we assume that x_t, y_t, z_t are represented in the natural binary number system (unsigned types in CUDA/C language). The transformation from the x_t, y_t, z_t node coordinates to the offset inside *data block* is defined by a linear mapping function $L(x_t, y_t, z_t)$. We use three linear mapping functions (all equations are valid for $a = 4$ nodes per tile edge):

$$L_{XYZ}(x_t, y_t, z_t) = x_t + 4 \cdot y_t + 4^2 \cdot z_t, \quad (11)$$

$$L_{YXZ}(x_t, y_t, z_t) = y_t + 4 \cdot x_t + 4^2 \cdot z_t \quad (12)$$

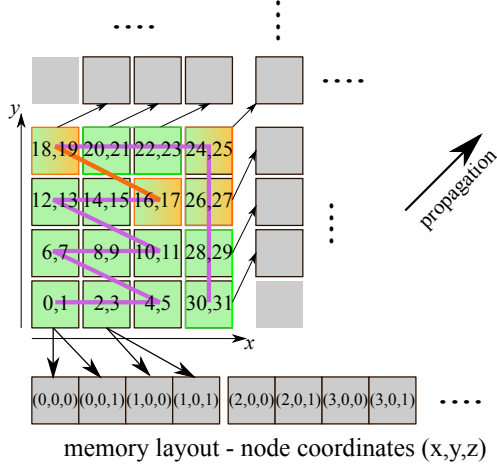


Figure 6: Propagation in north-east (NE) direction for zigzagNE memory layout and double precision f_i values. In this memory layout the two consecutive memory locations store f_i values for nodes with the same x and y coordinates - only z coordinate differs. Thus, each square on picture of tile denotes two 8-byte f_i values placed in neighbour memory locations. Partially utilised (uncoalesced) memory transactions (offsets 16 to 19 and 24 to 27) are marked orange. Each orange transaction is done twice and only half of data is used.

and

$$L_{zigzagNE}(x_t, y_t, z_t) = 2 \cdot \left(x_t + 3 \cdot y_t + \right. \\ \left. + ((x_t + 1) \cap 4) \cdot (3 - y_t) \right) + \quad (13) \\ \left. + (z_t \cap 1) + 4^2 \cdot (z_t \cap 2), \right.$$

where \cap denotes bitwise AND.

All mapping functions allow to minimise the number of uncoalesced memory transaction for our thread mapping and tile size. In our implementation we have started from the L_{XYZ} layout for all f_i data blocks and, after code profiling, we have modified layouts for these f_i data blocks, for which the redundant memory traffic was observed. L_{XYZ} have been used for $f_O, f_N, f_S, f_T, f_B, f_{NT}, f_{NB}, f_{ST}, f_{SB}, f_{LXZ}$ have been used for $f_E, f_W, f_{ET}, f_{EB}, f_{NW}, f_{SW}, f_{WT}, f_{WB}$ and $L_{zigzagNE}$ have been used for f_{NE} and f_{SE} . For this mapping functions assignment only for four f_i functions ($f_{NE}, f_{SE}, f_{NW}, f_{SW}$) the number of memory transactions is not minimal ($64 [nodes] \cdot 8 [B] / 32 [B/transaction] = 16$ transactions per f_i per tile). Both f_{NE} and f_{SE} require four additional memory transactions per tile (two per warp - marked orange in Fig. 6). Propagation of f_{NW} and f_{SW} requires 32 transactions per tile each, what gives 100% overhead. For f_{NW} and f_{SW} we also tried to use a mapping function similar to $L_{zigzagNE}$, but it resulted in slightly decreased performance despite the lowered number of memory transactions.

The total number of memory transactions required

for propagation of a tile is then $15 [f_i \text{ values}] \cdot 16 [transactions] + 2 [f_i \text{ values}] \cdot (16 + 4) [transactions] + 2 [f_i \text{ values}] \cdot 32 [transactions] = 344$ transactions. This gives a 13% overhead compared with the minimal number of transactions ($19 \cdot 16 = 304$).

3.3. Tiling overhead

Though tiles allow to precisely control data placement in memory, their fixed size can result in a low tile utilisation for sparse/irregular geometries. Compared with the minimal requirements defined in Sec. 2.3, the tiling brings some memory, bandwidth and computational overhead. The three main reasons for the tiling overhead are additional solid nodes inside tiles, two copies of f_i values, and some other data (e.g. tile bitmap). Furthermore, the data placement can cause an additional bandwidth overhead as shown in Sec. 3.2.

Let the overhead Δ be defined as a ratio of additional memory/operations (Δ^M - memory overhead, Δ^B - bandwidth overhead, Δ^C - computational overhead) to the minimum numbers defined in Sec. 2.3. Only one LBM time iteration is analysed since all iterations are processed in the same way. To simplify the equations, we use average values per tile, which enables us to omit multiplications by the number of tiles.

Let a denote the number of nodes per tile edge. Then, the number of nodes per tile is $n_{tn} = a^3$. If n_{tsn} denotes the average number of solid nodes per tile (we count only tiles containing non-solid nodes), and $n_{tfn} = n_{tn} - n_{tsn}$ denotes the average number of non-solid nodes per tile, then we can define the average tile utilization factor

$$\eta_t = \frac{n_{tfn}}{n_{tn}} = \frac{n_{tn} - n_{tsn}}{n_{tn}}. \quad (14)$$

When we assume that for each node the number of operations and used memory are minimal (i.e. that the tiling does not have an impact on the computational complexity for single nodes), then the overhead caused by low η_t results only from additional operations/memory, which must be done/allocated for all solid nodes from tile. In this case, the generic overhead value is then

$$\Delta_{\eta_t} = \frac{n_{tsn}}{n_{tfn}} = \frac{n_{tsn}}{n_{tn} - n_{tsn}} = \frac{1 - \eta_t}{\eta_t}. \quad (15)$$

In our implementation, the real values of bandwidth and computational overheads are even lower because we conditionally skip operations for solid nodes. For computational overhead, some operations are not done when the full warp of threads processes only solid nodes. This situation is rather rare (but possible), as a half of a tile must contain solid nodes. For bandwidth overhead the transfers are omitted, when values for solid nodes form a single 32-byte memory transaction. This happens quite often due to a spatial locality of geometry. Also, some transfers can be skipped, when during the propagation step the neighbour tile contains solid nodes.

For the memory overhead, Eqn. (15) allows only to find the overhead compared to an implementation where the two copies of f_i values are stored for each node. However, if we use as a reference the value defined by Eqn. (9), then the overhead must take into account the second copy of f_i

$$\begin{aligned}\Delta_{\eta_t}^M &= \frac{n_{tn} \cdot (2 \cdot q \cdot n_d + n_t) - n_{tfn} \cdot (q \cdot n_d + n_t)}{n_{tfn} \cdot (q \cdot n_d + n_t)} = \\ &= \frac{2 \cdot q \cdot n_d + n_t}{\eta_t \cdot (q \cdot n_d + n_t)} - 1 \approx \frac{2 - \eta_t}{\eta_t}.\end{aligned}\quad (16)$$

The final approximation is true for $n_t \ll q \cdot n_d$. Thus, the memory overhead compared with minimal requirements defined by Eqn. (9) grows about twice as fast as the overhead defined by Eqn. (15).

In addition to Eqn. (15) and (16), the memory and bandwidth overheads are also affected by the tile bitmap, but it usually may be skipped. For each non-empty tile, 3^3 values from the tile bitmap are read (neighbour tiles), and for each tile (both empty and non-empty), a single value must be stored in the tile bitmap. Each value in the tile bitmap can be at most 4-bytes long because 2^{32} tiles require much more memory, than is available in current GPUs (for D3Q19 lattice, double precision data and tile containing 4^3 nodes the data for all nodes require about $2^{32} \cdot 4^3 \cdot M_{node} \approx 39$ TiB of memory). Single tile requires $4^3 \cdot M_{node} \approx 9800$ B, what is almost 2500 times larger than the index in the tile bitmap. Thus, except for exceptionally sparse geometries (consisting significantly less than 1% of non-empty tiles) the tile bitmap overhead is negligible.

It can be seen from Eqn. (15) and (16) that the overhead grows quickly when the tile utilisation factor is lower than 1 (the overhead is 0.2 already for $\eta_t = 0.83$). Therefore, it is critical to achieve the tile utilisation as high as possible.

In Fig. 7 and 9 we show, how η_t changes for tiling of infinitely long channels running along one of the axes. Notice that the tile positions are discrete, thus only a few values of η_t can occur. It can be seen that tile utilisation above 0.8 can be always achieved for channels with diameter/side at least about 40 nodes. To have a guarantee that the tile utilisation is always above 0.9, the channels must have about 100 nodes across. Though these channel dimensions can be too large for some applications, it should be noted that these values are pessimistic. It is possible to find a tile placement that gives good tile utilisation even for very small channels (for example, η_t can be equal to 1 for a square channel as small as 4×4 nodes).

Additionally, since channels often are not parallel to the axes, then the values of η_t for real cases can be closer to the average value for different tilings of the same channel size (see Fig. 8). The average value for different tilings of the same channel size (the green line in Fig. 7 and 9) exceeds $\eta_t = 0.8$ for square channel 25^2 nodes and circular channel with diameter 30 nodes. For channel dimensions about 55 and 65 nodes for square and circular channels η_t exceeds 0.9.

Some observations can be also extracted from Fig. 7 and 9. It can be seen that the dispersion of the tile utilisation

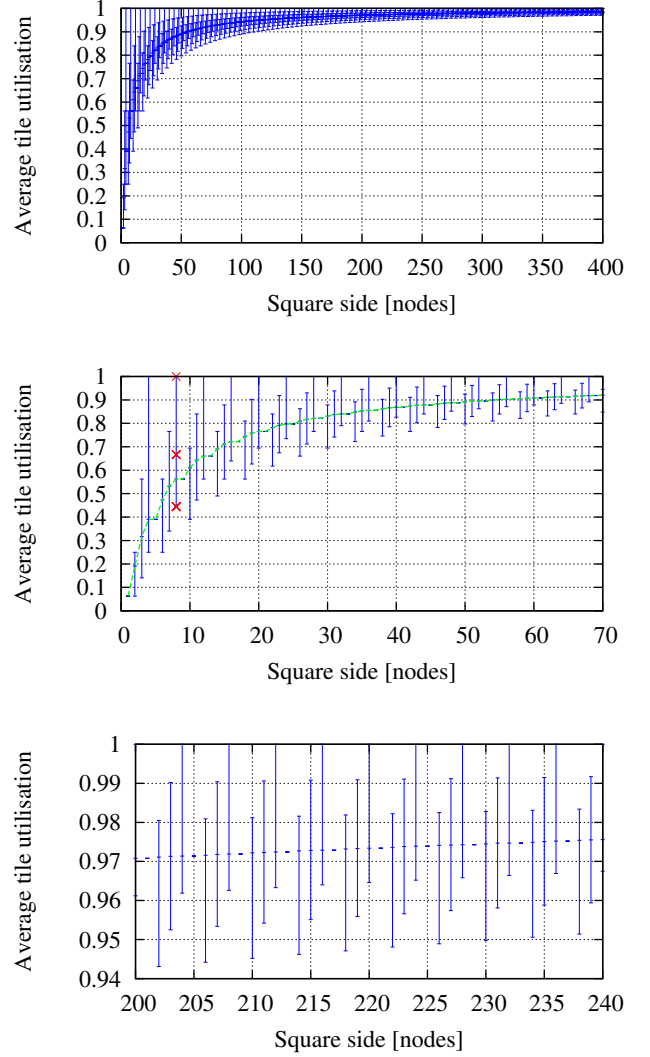


Figure 7: Average tile utilization for all available tilings of infinitely long square channel along the axis. For each channel size sixteen tilings are possible (see Fig. 8). Green line denotes the average value of average tile utilisation for given channel dimension (see Fig. 8). Red crosses denote real tile utilization for channel 8×8 nodes - there are only 3 available values.

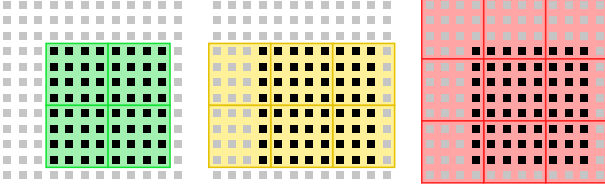


Figure 8: Cross section of 3 example tilings (marked on Fig. 7) of a square channel 8×8 nodes. Black/grey squares denote fluid/solid nodes. The channel is parallel to one of the axes. Average tile utilisation is $64/(4 \cdot 16) = 1.0$, $64/(6 \cdot 16) \approx 0.67$ and $64/(9 \cdot 64) \approx 0.44$ for green, yellow and red tiling. For this channel and tile size there are 9 tilings with average tile utilisation 0.44, 6 tilings similar to yellow tiling and only one green tiling with tile utilisation equal to 1. The average value of average tile utilisation is then $(9 \cdot 0.44 + 6 \cdot 0.67 + 1)/16 \approx 0.56$.

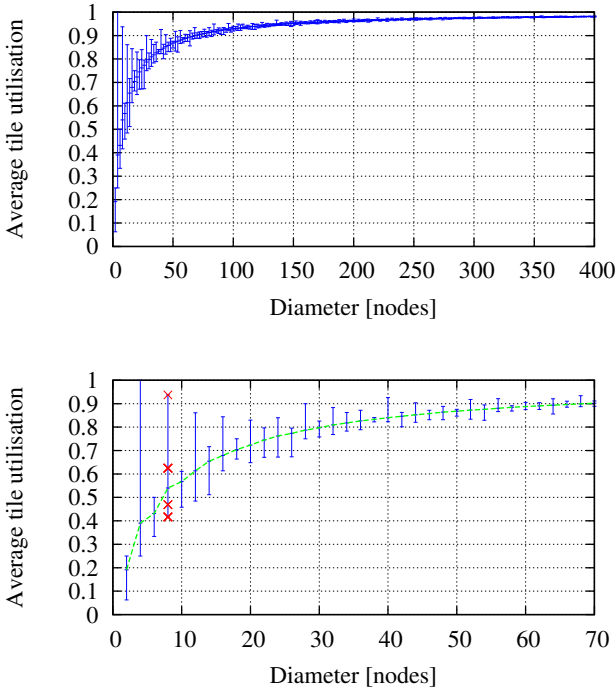


Figure 9: Average tile utilisation for all available tilings of infinitely long circular channel along the axis (16 different tilings are possible for each diameter). Green line denotes the average value of average tile utilisation for given channel size (see Fig. 8). Red crosses denote real tile utilization for channel with 8 nodes diameter - in this case only four possible values can occur.

for different tilings of a channel with given dimension is higher for square channels than for circular channels. It is possible to achieve higher tile utilisation for square channels, but it is also possible that the tile utilisation will be much lower than for circular ones.

The average value of different tile utilisations is higher for circular channels. The difference between the tile utilisation for circular and square channels depends greatly on the channel dimension: starting from 3 times for the smallest channels, through the 10% for channels with 20 nodes at diameter/side, and to the less than 2% for channels with at least 100 nodes.

For square channels, the two additional phenomena occur. First, if the channel dimension is larger by one node than the tile edge, then all tilings have the same tile utilisation. In this case, all tilings require the same number of tiles. The one node outside tile borders can always be placed in either "left" or "right" side of the tile, never on both sides. When the channel dimension is larger by two or more nodes than the tile edge, the number of additional tiles may differ depending on the tile placement. For example, two additional nodes can be put either in one additional tile or in two additional tiles on both sides. This behaviour can be used during geometry preparation to guarantee constant tile utilisation.

Second, the tile utilisation fluctuates with a period equal to the tile edge $a = 4$ nodes, i.e. for channel edge equal to $k \cdot a + m$, where $m \in \{0, 1, \dots, a - 1\}$ and integer $k \geq 0$. This results from the fact that for a single period (for the same k) the same numbers of tiles are needed: k , $k + 1$ or $k + 2$ tiles along each axis. Since the number of non-solid nodes grows with m , then for the same number of tiles the tile utilisation η_t becomes larger. As the lowest tile utilisation is always for $m = 2$, square channel side should be different from $k \cdot 4 + 2$.

3.4. Implementation details

Our code is written in CUDA C language ver. 7.5, the first version with official support for many C++11 features. Thanks to this we could write a highly templated code, which can be shared between the CPU and GPU compilers. The code sharing enabled us to simplify testing because results of many parts of the code could be thoroughly checked only on the CPU side. Additionally, extensive use of C++ templates allowed us to write a generic kernel code, which later can be specialised for fluid and collision models, data layouts, data precision, enabled optimisations etc. We were then able to test many combinations of the above parameters with minimal changes to source code and without code duplication. The disadvantages of this solution are a long compilation time and a large size of the executable file containing many versions of the same kernel.

In our implementation, the computations for a single LBM time step include collision, propagation and boundary computations for all nodes within the tiles. To minimise memory transfers, we combine collision, propagation

and boundary computations for a single node into a single GPU kernel. The kernel requires only one read of data from memory at the beginning, and one store of data at the kernel end. Local copies of data are stored in registers and in the shared memory. Separate tiles are processed by a separate GPU thread blocks (see Fig. 2), what allows for effortless synchronisation of computations within the tile. The general structure of our GPU kernel is shown in Fig. 10.

```

1: load copy of tile bitmap           ▷  $3^3$  values
2: load node types from current tile  ▷  $4^3$  values
3: load node types from neighbour tiles ▷ WLP
4: BARRIER
5: if node not solid then
6:   load  $f_0$ 
7:   for  $q \in \{1..18\}$  do
8:     compute address of neighbour node in direction  $q$ 
9:     if neighbour node not solid then
10:      gather  $f_q$  from neighbour node
11:    end if
12:  end for           ▷ all  $f_i$  copied to registers
13:  process boundary   ▷ calculate  $v, \rho$ 
14:  collide
15:  store all  $f_i$      ▷ all coalesced
16: end if

```

Figure 10: Structure of the GPU kernel implementing single LBM time iteration for a single node.

The propagation is implemented as a gathering data from neighbour nodes. The propagation between two nodes is done only when the target and source nodes are not solid. To minimise a cost of checking node types, we use shared memory to store copies of node types from current and neighbour tiles (lines 2–3 in Fig. 10). In code responsible for loading node types from neighbour tiles (line 3) we use warp level programming to balance the load of warps assigned to the tile. We also make a copy of 3^3 tile indices from the tile bitmap array forming a cube surrounding currently processed tile (line 1). All later operations requiring information about node and/or tile types use these copies.

After loading of node and tile types the barrier is needed (line 4 in Fig. 10), because in a later part of the kernel the threads from different warps require the data gathered in other warps. This is the only synchronisation point in the kernel.

All later operations are done only when a node assigned to current thread is not solid (line 5). In this way, we can skip unnecessary operations as mentioned in Sec. 3.3.

The computations for a non-solid node are divided into the three parts: lines 6–12 are responsible for gathering f_i values, lines 13 and 14 perform all floating point calculations and line 15 stores the computed f_i values to the memory. All operations are implemented as standard except for

the line 8, where we compute indices for a neighbour node. First, the index of a tile containing the neighbour node is computed - we use only the values from $\{-1, 0, 1\}$ due to local copy of the tile bitmap. When the neighbour tile is not empty, a few indices for the neighbour node are computed since we need both information about the neighbour node type (this is gathered from the copy in shared memory) and the value of proper f_i (which is stored in memory). Notice that for different f_i functions we need to use the different data layouts (see Sec. 3.2).

In the GPU kernel, we have also applied some optimisations not shown in Fig. 10. First, the loop starting at line 7 is partially unrolled to allow computations for two f_i functions in a single iteration, what increases the instruction level parallelism due to the interlacing of two independent streams of instructions. In addition, the order of iterations over q is chosen in a way that allows to share some parts of neighbour node address computations (line 8). Next, some values (e.g. v) are stored in shared memory to minimise register pressure. Shared memory is also used in the implementation of the bounce back boundary condition to avoid a significant increase of required registers. To minimise conflicts in shared memory, 64-bit mode is used. We have also tuned the usage of registers for each specialisation of the kernel since different collision models require different amounts of temporary data. The cost of divergence in the code responsible for computations of an address of the neighbour node is minimised - only integer indices are computed in divergent branches, time-consuming operations like memory accesses are done in all threads simultaneously. In places, where it is possible and useful, the number of costly divisions is reduced by replacing them with multiplications by the inverse. Finally, we have been intensively using inline methods and compiler pragmas to force loop unrolling, what allowed us to connect clean, structured code with high performance.

4. Results

4.1. Measurements methodology

All tests have been run on a computer with the Intel i7-3930K CPU and 64 GB quad-channel DDR3 DRAM. The installed GPU was GTX Titan (Kepler architecture) clocked at 823 MHz with 6 GB GDDR5 memory clocked at 3.004 GHz. The peak theoretical GPU memory bandwidth is 288.384 GB/s (GB/s denotes 10^9 B/s), the NVIDIA *bandwidthTest* utility reported 228.5 GB/s (79.2% of peak). We have been using Linux operating system with NVIDIA CUDA Toolkit 7.5 installed.

To identify performance limits we have been using three versions of kernels: kernels implementing all LBM operations, kernel with full propagation but without computations, and kernel without propagation, which only reads and writes node data for the same node (without communication between neighbour nodes). The kernels implementing all LBM operations are described by collision and

fluid models (e.g. "BGK incompressible"), the kernel with propagation is marked as "propagation only" and the last kernel without full propagation is denoted as "read/write only". All kernels operate on double precision floating point numbers.

As a measure of kernel performance we have been using the number of $10^6 \times$ non-solid (fluid and boundary) nodes processed per second (MLUPS). The processing time was measured for separate kernel runs (with necessary synchronisation). For each kernel and case we run 1000 steps and measured wall clock time for each kernel run. The first few (1-3) kernel runs usually took more time than the rest, thus before processing we removed the first five measurements. For time measurement we have been using the C++ *std::chrono* library with microsecond resolution.

4.2. Performance for cavity3D

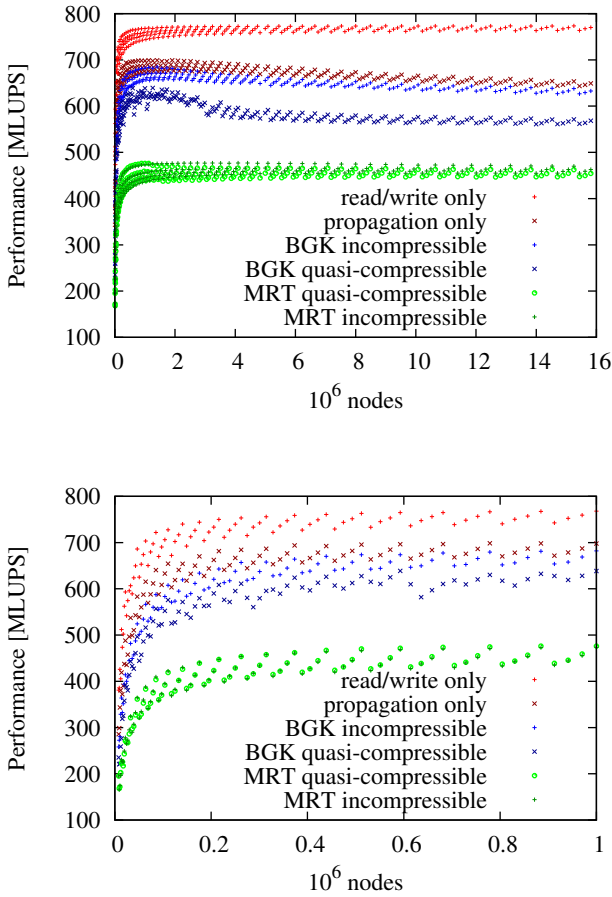


Figure 11: Kernel performance for cavity3D with geometry size b^3 nodes, where $b \in \langle 20, 252 \rangle$. All measures on GTX Titan.

To identify performance limits not affected by the geometry sparsity, we have first measured the performance of our implementation for the cavity3D test case, a standard example of dense geometry. The performance of all

kernels as a function of a geometry size is shown in Fig. 11.

The performance dependency on the geometry size is similar to many other GPU implementations. For 10^5 nodes the kernels achieve about 85% of their maximum performance. To achieve more than 95% of maximum performance, at least 3×10^5 nodes are needed. For other GPUs the numbers of nodes can be different, but for today's machines the maximum performance should always be possible to achieve for geometries containing about 10^6 or more nodes.

The highest performance was observed for read/write only kernel: 773 MLUPS what corresponds to memory bandwidth equal to 236.5 GB/s (82% of maximal theoretical bandwidth for GTX Titan). Because in this kernel all memory transactions are coalesced, we achieved bandwidth utilisation even higher than reported by the *bandwidthTest* utility.

For kernel with propagation only the performance dropped by 9.6 % to 699 MLUPS resulting in bandwidth 213.9 GB/s (74.2% of maximum). It should be noted that though the propagation only kernel has more than seven times more instructions than read/write only kernel (711 vs 99 instructions), the performance penalty caused by neighbour tile data access during propagation is smaller than 10%. More than 60% of all instructions in propagation only kernel are integer arithmetic operations used for address computations.

Finally, after enabling all computations the kernel performance dropped by an additional 2.3% for BGK incompressible (684 MLUPS, 72.6% of GPU memory bandwidth) up to 32% for MRT quasi-compressible (476 MLUPS, 50.5% of GPU memory bandwidth). The large performance drop for MRT results from computational overhead - due to the large register usage per single thread and the low GPU occupancy the complex double precision computations are not masked with memory transfers.

The performance of our implementation for sparse geometries is close even to the highly optimised version for dense geometries from [1], which utilises 76.7% of the maximum memory bandwidth for Tesla K20c and the BGK quasi-compressible model. Moreover, the implementation from [1] uses single precision numbers what allows to increase GPU utilisation due to much lower register pressure. This result shows that it is possible to process sparse geometries with performance similar to dense ones.

As can be seen in Fig. 11, the performance of all kernel versions fluctuates for geometry edge $b = 4 \cdot k + m$, where $m \in \{0, \dots, 3\}$. This results from the low tile utilisation for tiles placed on the geometry edges (see Section 3.3). Notice that a performance difference for the geometry edge b computed for the same k and different m decreases for larger k , e.g. for $b = 100$ the performance of propagation only kernel is between 672 and 699 MLUPS, and for $b = 248$ between 643 and 653 MLUPS. This results primarily from differences in the average tile utilisation: for the cubic cavity3D geometry the number of fully utilised tiles grows

Table 3: Kernel performance in MLUPS for dense geometry cavity3D. *Max perf.* denotes the performance for a case with the highest performance, *b* is the dimension of the case with the highest performance (number of nodes is b^3), *Max case* denotes the performance for the largest case (252^3 nodes).

Operation	Max perf.	<i>b</i>	Max case
read/write only	773	208	773
propagation only	699	104	653
BGK incompressible	684	112	637
BGK quasi-compressible	639	108	571
MRT incompressible	477	148	470
MRT quasi-compressible	476	104	461

proportionally to b^3 and the number of partially utilised tiles proportionally to b^2 , what causes the growth of the average tile utilisation for larger b .

Some interesting behaviour is also shown in Fig. 11 - after achieving peak about 10^6 nodes the performance slowly decreases with a geometry size increase. The intensity of this phenomenon depends on a computational complexity of operation - the most visible is for propagation only (see Table 3). Also, it does not occur for the kernel without propagation (read/write only). This is a result of data propagation between neighbour tiles that increases the time needed for tile processing - for read/write only kernel there is no propagation between tiles, thus no performance decrease is observed. For smaller geometries the ratio of tile faces and edges common to two tiles to the number of tiles is smaller than for large geometries, thus the performance for smaller geometries is higher.

4.3. Propagation performance

To estimate how the communication between tiles affects the performance we have also measured the performance of propagation for a set of rectangular channels containing about 10^6 nodes. The channels differ in dimensions: we have used channels starting from $4 \times 4 \times 62500$ nodes up to $100 \times 100 \times 100$ nodes (1873 combinations in total). This way allowed us to generate geometries with a different number of faces and edges common to neighbour tiles.

Let η_f denote the number of common faces per tile and η_e denote the number of common edges per tile. The values of η_f and η_e are computed based on the geometric layout, e.g. for four tiles arranged in $2 \times 2 \times 1$ mesh there are 4 common faces and 1 common edge, thus $\eta_f = 4/4$ and $\eta_e = 1/4$. For channel dimensions used in our measurements the values of η_f and η_e are the following: channel $4 \times 4 \times 62500$ had about one common face per tile and zero edges, channel $4 \times 8 \times 31248$ had about 1.5 common face per tile and about 0.5 common edge per tile, and so forth. We do not count the numbers of transferred faces/edges because for rectangular channels each face is always common between two tiles and each edge between four, thus

the number of data transfers is proportional to η_f and η_e .

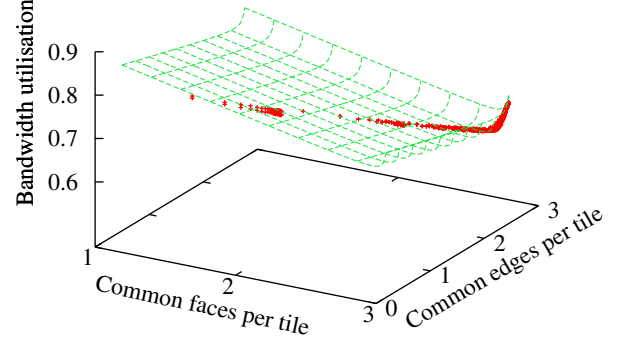


Figure 12: Performance of propagation as a function of a number of common faces and edges per tile. The propagation performance is shown as a fraction of the full GPU memory bandwidth. The measurements have been done for rectangular channels containing 10^6 nodes but with different dimensions.

The propagation performance as a function of common faces per tile and common edges per tile ratios is shown in Fig. 12. The highest performance (851 MLUPS, 90% GPU memory bandwidth utilisation) was observed for channel $4 \times 62500 \times 4$ nodes.

Most of the points shown in Fig. 12 is placed close to a plane parallel to the "Bandwidth utilisation" axis and defined by a line

$$\eta_e = 1.85 \cdot \eta_f - 2.56, \quad (17)$$

which can be treated as a coarse estimation of a ratio of common edges to faces. Only for small η_f ($\eta_f < 2$), which corresponds to very narrow channels of dimension $1 \times k$ tiles, the line defining the plane becomes

$$\eta_e = \eta_f - 1. \quad (18)$$

The bandwidth utilisation shown in Fig. 12 can be estimated as

$$BU_p = 0.92 - \frac{\eta_f}{14.28} - \frac{\eta_e}{25.74} + \frac{0.00104}{2.9 - \eta_f} + \frac{0.0023}{2.81 - \eta_e}. \quad (19)$$

Two main areas can be identified in the dependence shown in Eqn. (19). For $\eta_f < 2.8$ and $\eta_e < 2.6$ the propagation performance falls proportionally to η_f and η_e . Also, the η_f is almost twice as important as η_e . For $\eta_f < 2.8$ and $\eta_e < 2.6$ the propagation performance increases in inverse proportion to η_f and η_e . In this range η_e is more important than η_f , what means that the more square channels (with width similar to depth) allow to achieve a higher performance. The highest bandwidth utilisation in this range was observed for channel $100 \times 100 \times 100$ nodes (74%, 698 MLUPS).

Table 4: Propagation performance in MLUPS for different data layouts inside tile. All values for cavity3D with 100^3 nodes (25^3 tiles).

Data layout	Performance	BW utilisation	L2 reads	Memory reads	Cache hit rate	Instructions
XYZ	586	0.622	8 188 609	7 298 421	0.109	673
XYZ + zigzagNE	606	0.643	7 827 219	6 824 451	0.128	701
XYZ + YXZ	666	0.707	6 730 769	5 397 210	0.198	681
all three	698	0.741	6 368 191	4 919 420	0.228	711
rw only	768	0.815	4 812 588	4 812 524	$\approx 10^{-5}$	99

4.4. Data layout impact

To measure the performance gains caused by tile memory layouts presented in Sec. 3.2 we have used *nvprof* utility to profile the four propagation kernels with memory layouts XYZ, XYZ + zigzagNE, XYZ + YXZ, and all three (XYZ + YXZ + zigzagNE). The assignment of memory layouts to f_i data blocks is defined in Sec. 3.2. For cases XYZ + zigzagNE and XYZ + YXZ the default memory layout is XYZ. The results are shown in Table 4.

As a test case, we have used the geometry containing $100 \times 100 \times 100$ nodes due to its regular structure that allows to exactly compute the number of transactions. The minimal number of write transactions is $25^3 [\text{tiles}] \cdot 19 [f_i] \cdot 16 [32\text{-byte transactions}] = 4\,750\,000$ 32-byte writes. The minimal number of 32-byte reads is 4 750 000 plus transactions for node type reads: $15625 \cdot 64 \cdot 2 / 32 = 62\,500$ 32-byte transactions. The total number of memory transactions is then $4\,750\,000 + 4\,812\,500 = 9\,562\,500$.

The measured numbers of 32-byte memory writes were independent of the used memory layout. For all layouts we have observed 4 750 000 32-byte writes, which is the minimal value. This is a result of the *gather* pattern, where irregular memory accesses to neighbour nodes/tiles occur only during the read stage.

The data from Table 4 show that the proposed memory layout ("all three" row) allows to achieve the number of memory transactions (both read and write) only 1.12% larger than the minimal. This is a result of both a reduction of uncoalesced memory reads (L2 reads column) and an increase of cache hit rate. Notice also that for "rw only" kernel the numbers of cache and device memory transactions is minimal, what shows the correctness of our theoretical estimations of numbers of memory transactions.

Since the YXZ layout is applied to the largest number of f_i data blocks (eight), its use results in the largest performance increase (13.65% compared with 586 MLUPS for XYZ layout). The zigzagNE layout is used only for two f_i data blocks and gives exactly four times smaller performance increase (3.41%). Notice that the performance increase does not depend on address computation complexity. Though zigzagNE layout requires 14 times more additional instructions per f_i data block than YXZ layout (14 compared with 1 instruction per f_i data block), the performance increase scales linearly only with the number of used data blocks regardless of the address computation

complexity.

The reductions of the numbers of transactions (for both cache and device memories) for the zigzagNE and YXZ layouts add up: $(8188609 - 7827219) + (8188609 - 6730769) \approx (8188609 - 6368191)$ and $(7298421 - 6824451) + (7298421 - 5397210) \approx (7298421 - 4919420)$. This means that despite the parallel execution of kernels for 14 [*multiprocessors*] \cdot 16 [*tiles*] and shared L2 cache and device memories there is no visible dependence between accesses to different f_i data blocks.

4.5. Performance for sparse geometries

Table 5: Kernels performance in MLUPS for arrays of random arranged spheres with porosities equal to 70, 80 and 90 percent (similar to [2]). Geometry size is 192^3 nodes. Results from [2] are for Tesla K20 GPU and kernel similar to our BGK incompressible.

Kernel	70%	80%	90%
read/write only	671	637	574
propagation only	600	561	494
BGK incompressible	566	525	452
BGK quasi-compressible	518	477	407
MRT incompressible	348	321	281
MRT quasi-compressible	351	321	280
[2]	334	330	337

To verify the performance of our implementation for sparse geometries we have run simulations for cases similar to presented in [2] and [4]. In Table 5 there are shown results for three arrays of random arranged spheres with a diameter of 40 lattice units. Compared with implementation of the same collision model from [2] our solution achieves the higher bandwidth utilisation for cases with porosity 70% and 80% (respectively 60% and 55.7% of peak memory bandwidth versus 49.1% and 48.6% for [2]). For porosity 90% our implementation is slightly worse (48% vs 49.6% of peak memory bandwidth). Average tile utilisation factors for these three cases are 0.699, 0.584 and 0.512.

The second test was the blood flow in cerebral aneurysm (see Fig. 13). We used the model similar to presented in [2], but since our implementation does not support multi-GPU configurations, we reduced the resolution to $730 \times 342 \times 334$ nodes (84.6×10^6 nodes in total, 14.8×10^6

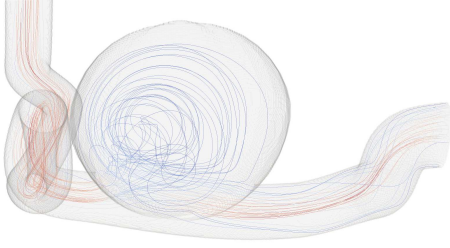


Figure 13: Flow pattern within the cerebral aneurysm model.

non-solid nodes, geometry porosity 0.825, average tile utilisation $\eta_t = 0.931$). The results are shown in Table 6.

For the cerebral aneurysm case our implementation achieves significantly higher GPU memory bandwidth utilisation than [2]. This is due to the high tile utilisation resulting from a good spatial locality of geometry. Notice also that the numbers from Table 6 are close to the values achieved for dense geometries (the right column in Table 3). It shows that when the tile utilisation is high, then the performance of our implementation is almost not dependent on geometry sparsity.

Table 6: Kernels performance for cerebral aneurysm model similar to [2]. Results from [2] are for single precision floating point numbers, four Tesla C1060 GPUs and kernel similar to our BGK incompressible.

Kernel	MLUPS	BW utilisation
read/write only	755	0.801
propagation only	656	0.696
BGK incompressible	638	0.677
BGK quasi-compressible	572	0.607
MRT incompressible	447	0.474
MRT quasi-compressible	437	0.464
[2]	1090	0.410

Finally, we have measured the performance for aorta with coarctation case (see Fig. 14) similar to presented in [4]. The performance comparison is shown in Table 7. Since this geometry is rather small (resolution $93 \times 161 \times 442$, about 0.66×10^6 non-solid nodes, porosity 0.906), the performance of our implementation is up to 14% lower than for the cerebral aneurysm case shown in Table 6. Small geometry size degrades performance not only due to the low number of nodes (as shown in Fig. 11) but primarily due to the reduced tile utilisation (equal to 0.807 in this case). Despite this performance degradation, the tile based solution achieves significantly higher memory bandwidth utilisation than implementation presented in [4].

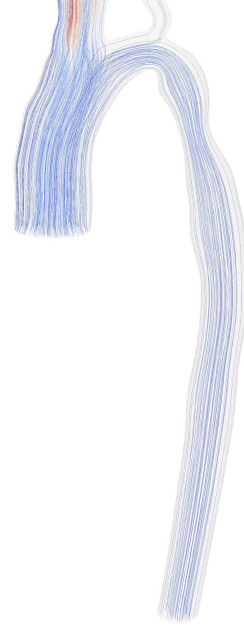


Figure 14: Flow pattern within the model of aorta with coarctation.

Table 7: Kernels performance for aorta model with coarctation (similar to [4]). Results from [4] are for GTX 680 GPU, D3Q15 lattice and kernel similar to our BGK quasi-compressible. The values in last row are estimated based on results presented in [4].

Kernel	MLUPS	BW utilisation
read/write only	717	0.761
propagation only	649	0.689
BGK incompressible	621	0.659
BGK quasi-compressible	574	0.609
MRT incompressible	386	0.410
MRT quasi-compressible	384	0.407
[4]	~ 150	~ 0.2

5. Conclusions

In this paper, the GPU based LBM implementation for fluid flows in sparse geometries is presented. In contrary to the previous solutions for sparse geometries, which use the indirect node addressing, our implementation covers the geometry with the uniform mesh of cubic tiles. Due to the fixed tile size, we have been able to carefully arrange data inside a tile what allowed to significantly decrease the GPU memory traffic. The proposed data layout allowed us to achieve up to 684 MLUPS (72.6% utilisation of maximum theoretical memory bandwidth) on GTX Titan for D3Q19 lattice, BGK incompressible model and double precision data. We have also examined the performance of our implementation for both BGK and MRT collision models in incompressible and quasi-compressible versions. The performance comparison with existing implementations shows that in most real cases our solution allows to achieve a higher performance. Especially promising results were obtained for the large geometry with good spatial locality (cerebral aneurysm model with 14.8×10^6 non-solid nodes), where we observed up to $1.65\times$ higher memory bandwidth utilisation than the highly optimised implementation based on indirect node addressing. Only for the very sparse geometries with weak spatial locality (array of random arranged spheres with porosity 0.9) our implementation is slightly slower (48% vs 49.6% of peak memory bandwidth) due to a low tile utilisation.

Future work includes the extension to a multi-GPU version to increase the size of supported geometries and the search for a method to increase the tile utilisation (e.g. by a non-uniform tile placement and/or a decrease of the tile size).

Acknowledgments

This work was supported by Wrocław University of Technology, Faculty of Electronics, Chair of Computer Engineering [the statutory grant No. 50228].

References

References

- [1] M. J. Mawson, A. J. Revell, Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs, *Computer Physics Communications* 185 (10) (2014) 2566–2574.
- [2] C. Huang, B. Shi, Z. Guo, Z. Chai, Multi-GPU based lattice Boltzmann method for hemodynamic simulation in patient-specific cerebral aneurysm, *Communications in Computational Physics* 17 (2015) 960–974.
- [3] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, E. Kaxiras, A flexible high-performance lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries, *Concurr. Comput. : Pract. Exper.* 22 (1) (2010) 1–14.
- [4] C. Nita, L. Itu, C. Suci, GPU accelerated blood flow computation using the lattice Boltzmann method, in: *High Performance Extreme Computing Conference (HPEC)*, 2013 IEEE, 2013, pp. 1–6.
- [5] NVIDIA Corporation, *CUDA C Programming Guide PG-02829-001_v7.5*, 2015.

- [6] P. L. Bhatnagar, E. P. Gross, M. Krook, A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems, *Physical Review* 94 (3) (1954) 511–525.
- [7] Q. Zou, S. Hou, S. Chen, G. D. Doolen, A improved incompressible lattice Boltzmann model for time-independent flows, *Journal of Statistical Physics* 81 (1-2) (1995) 35–48.
- [8] X. He, L.-S. Luo, Lattice Boltzmann model for the incompressible Navier–Stokes equation, *Journal of Statistical Physics* 88 (3-4) (1997) 927–944.
- [9] P. Lallemand, L.-S. Luo, Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability, *Physical Review E* 61 (6) (2000) 6546–6562.
- [10] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, U. Rude, Optimization and profiling of the cache performance of parallel lattice Boltzmann codes, *Parallel Processing Letters* 13 (04) (2003) 549–560.
- [11] G. Wellein, T. Zeiser, G. Hager, S. Donath, On the single processor performance of simple lattice Boltzmann kernels, *Comput. Fluids* 35 (8-9) (2006) 910–919.